

Abstract Diagnosis for *tccp* using a Linear Temporal Logic

MARCO COMINI, LAURA TITOLO

DIMI, Università degli Studi di Udine, Italy
(e-mail: marco.comini@uniud.it)

ALICIA VILLANUEVA

DSIC, Universitat Politècnica de València, Spain
(e-mail: villanue@dsic.upv.es)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Automatic techniques for program verification usually suffer the well-known state explosion problem. Most of the classical approaches are based on browsing the structure of some form of model (which represents the behavior of the program) to check if a given specification is valid. This implies that a part of the model has to be built, and sometimes the needed fragment is quite huge.

In this work, we provide an alternative automatic decision method to check whether a given property, specified in a linear temporal logic, is *valid* w.r.t. a *tccp* program. Our proposal (based on abstract interpretation techniques) does not require to build any model at all. Our results guarantee correctness but, as usual when using an abstract semantics, completeness is lost.

KEYWORDS: concurrent constraint paradigm, linear temporal logic, abstract diagnosis, decision procedures, program verification

1 Introduction

The Concurrent Constraint Paradigm (*ccp*, (Saraswat 1989)) is a simple, logic model which is different from other (concurrent) programming paradigms mainly due to the notion of store-as-constraint that replaces the classical store-as-valuation model. It is based on an underlying constraint system that handles constraints on variables and deals with partial information. Within this family, (de Boer et al. 2000) introduced the *Timed Concurrent Constraint Language* (*tccp* in short) by adding to the original *ccp* model the notion of time and the ability to capture the absence of information. With these features, one can specify behaviors typical of reactive systems such as *timeouts* or *preemption* actions.

It is well-known that modeling and verifying concurrent systems by hand can be an extremely hard task. Thus, the development of automatic formal methods is essential. One of the most known techniques for formal verification is model checking, that was originally introduced in (Clarke and Emerson 1981; Queille and Sifakis 1982) to automatically check if a finite-state system satisfies a given property. It consisted in an exhaustive

analysis of the state-space of the system; thus the state-explosion problem is its main drawback and, for this reason, many proposals in the literature try to mitigate it.

All the proposals of model checking have in common that a *part* of the model of the (target) program has to be built, and sometimes the needed fragment is quite huge. In this work, we propose a completely different approach to the formal verification of temporal (LTL) properties of concurrent (reactive) systems specified in *tccp*. We formalize a method to validate a specification of the expected behavior of a *tccp* program P , expressed by a linear temporal formula ϕ , which does not require to build any model at all.

The linear temporal logic we use to express specifications, **csLTL**, is an adaptation of the propositional LTL logic to the concurrent constraint framework. This logic is also used as the basis of the abstract domain for a new (abstract) semantics for the language.

In brief, our method is an *extension* of abstract diagnosis for *tccp* (Comini et al. 2011) where the abstract domain \mathbb{F} is formed by **csLTL** formulas. We cannot use the original abstract diagnosis framework of (Comini et al. 2011) since \mathbb{F} is not a complete lattice.

The contributions of this work are the following:

- A new abstract semantics for *tccp* programs based on **csLTL** formulas;
- A novel and effective method to validate **csLTL** properties based on the ideas of abstract diagnosis. This proposal intuitively consists in viewing P as a formula transformer by means of an (abstract) immediate consequence operator $\hat{\mathcal{D}}[P]$ which works on **csLTL** formulas. Then, to decide the validity of ϕ , we just have to check if $\hat{\mathcal{D}}[P]_\phi$ (i.e., the P -transformation of ϕ) implies ϕ ;
- An automatic decision procedure for **csLTL** properties that makes our method effective.

With our technique we can check, for instance, that, at a railway crossing system, each time a train is approaching, the gate is down, or that whenever a train has crossed, the gate is up. When a property is non valid, the method identifies the buggy process declaration. Technical results of Sections 3 and 4 can be found in (Comini et al. 2014).

2 The small-step operational behavior of the *tccp* language

The *tccp* language (de Boer et al. 2000) is particularly suitable to specify reactive and time critical systems. As the other languages of the *ccp* paradigm (Saraswat 1993), it is parametric w.r.t. a *cylindric constraint system* which handles the data information of the program in terms of constraints. The computation progresses as the concurrent and asynchronous activity of several agents that can accumulate information in a *store*, or query information from it. Briefly, a cylindric constraint system¹ is an algebraic structure $\mathbf{C} = \langle \mathcal{C}, \leq, \otimes, \text{false}, \text{true}, \text{Var}, \exists \rangle$ composed of a set of constraints \mathcal{C} such that (\mathcal{C}, \leq) is a complete algebraic lattice where \otimes is the *lub* operator and *false* and *true* are respectively the greatest and the least element of \mathcal{C} ; *Var* is a denumerable set of variables and \exists existentially quantifies variables over constraints. The *entailment* \vdash is the inverse of \leq .

Given a cylindric constraint system \mathbf{C} and a set of process symbols Π , the syntax of agents is given by the grammar:

$$A ::= \text{skip} \mid \text{tell}(c) \mid A \parallel A \mid \exists x A \mid \sum_{i=1}^n \text{ask}(c_i) \rightarrow A \mid \text{now } c \text{ then } A \text{ else } A \mid p(\vec{x})$$

¹ See (de Boer et al. 2000; Saraswat 1993) for more details on cylindric constraint systems.

$$\begin{array}{c}
\frac{}{\langle \text{tell}(c), d \rangle \rightarrow \langle \text{skip}, c \otimes d \rangle} \quad d \neq \text{false} \\
\frac{\langle A, d \rangle \rightarrow \langle A', d' \rangle}{\langle \text{now } c \text{ then } A \text{ else } B, d \rangle \rightarrow \langle A', d' \rangle} \quad d \vdash c \\
\frac{\langle B, d \rangle \rightarrow \langle B', d' \rangle}{\langle \text{now } c \text{ then } A \text{ else } B, d \rangle \rightarrow \langle B', d' \rangle} \quad d \not\vdash c \\
\frac{\langle A, d \rangle \rightarrow \langle A', d' \rangle \quad \langle B, d \rangle \rightarrow \langle B', c' \rangle}{\langle A \parallel B, d \rangle \rightarrow \langle A' \parallel B', d' \otimes c' \rangle} \\
\frac{\langle A, l \otimes \exists_x d \rangle \rightarrow \langle B, l' \rangle}{\langle \exists^l x A, d \rangle \rightarrow \langle \exists^{l'} x B, d \otimes \exists_x l' \rangle}
\end{array}
\qquad
\begin{array}{c}
\frac{}{\langle \sum_{i=1}^n \text{ask}(c_i) \rightarrow A_i, d \rangle \rightarrow \langle A_j, d \rangle} \quad j \in [1, n], \quad d \vdash c_j, \quad d \neq \text{false} \\
\frac{\langle A, d \rangle \not\vdash}{\langle \text{now } c \text{ then } A \text{ else } B, d \rangle \rightarrow \langle A, d \rangle} \quad d \vdash c, \quad d \neq \text{false} \\
\frac{\langle B, d \rangle \not\vdash}{\langle \text{now } c \text{ then } A \text{ else } B, d \rangle \rightarrow \langle B, d \rangle} \quad d \not\vdash c \\
\frac{\langle A, d \rangle \rightarrow \langle A', d' \rangle \quad \langle B, d \rangle \not\vdash}{\langle A \parallel B, d \rangle \rightarrow \langle A' \parallel B, d' \rangle} \quad \langle B \parallel A, d \rangle \rightarrow \langle B \parallel A', d' \rangle \\
\frac{}{\langle p(\vec{x}), d \rangle \rightarrow \langle A, d \rangle} \quad p(\vec{x}) := A \in D, \quad d \neq \text{false}
\end{array}$$

Fig. 1. The transition system for *tccp*.⁴

where c, c_1, \dots, c_n are finite constraints in \mathbf{C} ; $p/m \in \Pi$ and \vec{x} denotes a generic tuple of m variables. A *tccp* program is an object of the form $D . A$, where A is an agent, called *initial agent*, and D is a set of *process declarations* of the form $p(\vec{x}) := A$ (for some agent A). The notion of time is introduced by defining a discrete and global clock.

The *operational semantics* of *tccp*, defined in (de Boer et al. 2000), is formally described by a transition system $T = (\text{Conf}, \rightarrow)$. Configurations in Conf are pairs $\langle A, c \rangle$ representing the agent A to be executed in the current global store c . The transition relation $\rightarrow \subseteq \text{Conf} \times \text{Conf}$ is the least relation satisfying the rules of Figure 1. Each transition step takes exactly one time-unit.

Example 2.1 (Guiding example) Through the paper, we use as guiding example a part of the full specification of a railway crossing system introduced in (Alpuente et al. 2006). Let us call D_m the following *tccp* declaration:

$$\begin{aligned}
\text{master}(C, G) := & \exists C', G' \big(\text{now } (C = [\text{near} \mid _]) \text{ then} \\
& \quad \text{tell}(C = [\text{near} \mid C']) \parallel \text{tell}(G = [\text{down} \mid G']) \parallel \text{master}(C', G') \\
& \text{else now } (C = [\text{out} \mid _]) \text{ then} \\
& \quad \text{tell}(C = [\text{out} \mid C']) \parallel \text{tell}(G = [\text{up} \mid G']) \parallel \text{master}(C', G') \\
& \text{else master}(C, G) \big)
\end{aligned}$$

Due to the monotonicity of the store, streams (written in a list-fashion way) are used to model *imperative-style* variables (de Boer et al. 2000). The master process uses an *input channel* C (implemented as a stream) through which it receives signals from the environment (trains), and an *output channel* G through which it sends orders to a gate process. It checks the input channel for a *near* signal (the guard in the first *now* agent), in which case it sends (tells) the order *down* through G , links the future values (C') of the stream C and restarts the check at the following time instant (recursive call $\text{master}(C', G')$). If the *near* signal is not detected, then, the *else* branch looks for the *out* signal and (if present) behaves dually to the first branch. Finally, if no signal is detected at the current time instant (last *else* branch), then the process keeps checking from the following time instant. ■

⁴ The auxiliary agent $\exists^l x A$ makes explicit the local store l of A . This auxiliary agent is linked to the principal hiding construct by setting the initial local store to *true*, thus $\exists x A := \exists^{\text{true}} x A$.

In this work, we prove the correctness of our technique w.r.t. the denotational concrete semantics of (Comini et al. 2013a), which is fully-abstract (correct and complete) w.r.t. the small-step operational behavior of *tccp*. Also **csLTL** is interpreted over this denotational model. We thus introduce the most relevant aspects of such semantics.

The denotational semantics of a *tccp* program consists of a set of *conditional (timed) traces* that represent, in a compact way, all the possible behaviors that the program can manifest when fed with an *input* (initial store). Conditional traces can be seen as hypothetical computations in which, for each time instant, we have a condition representing the information that the global store has to satisfy in order to proceed to the next time instant. Briefly, a conditional trace is a (possibly infinite) sequence $t_1 \cdots t_n \cdots$ of *conditional states*, which can be of three forms:

conditional store: a pair $\eta \rightarrow c$, where η is a *condition* and $c \in \mathbf{C}$ a store;
stuttering: the construct $stutt(C)$, with $C \subseteq \mathbf{C} \setminus \{true\}$;
end of a process: the construct \boxtimes .

Intuitively, the conditional store $\eta \rightarrow c$ means that, provided condition η is satisfied by the current store, the computation proceeds so that in the following time instant, the store is c . A *condition* η is a pair $\eta = (\eta^+, \eta^-)$ where $\eta^+ \in \mathbf{C}$ and $\eta^- \in \wp(\mathbf{C})$ are called positive and negative condition, respectively. The positive/negative condition represents information that a given store must/must not entail, thus they have to be consistent in the sense that $\forall c^- \in \eta^- \eta^+ \not\vdash c^-$. The stuttering construct models the suspension of the computation when none of the guards in a non-deterministic agent is satisfied. C is the set of guards in the non-deterministic agent. Conditional traces are monotone (i.e., for each $t_i = \eta_i \rightarrow c_i$ and $t_j = \eta_j \rightarrow c_j$ such that $j \geq i$, $c_j \vdash c_i$) and consistent (i.e., each store in a trace does not entail the negative conditions of the following conditional state).

We denote the domain of *conditional trace sets* as \mathbb{M} . $(\mathbb{M}, \sqsubseteq, \sqcup, \sqcap, \mathbf{M}, \{\epsilon\})$ is a complete lattice, where $M_1 \sqsubseteq M_2 \Leftrightarrow \forall r_1 \in M_1 \exists r_2 \in M_2. r_1$ is a prefix of r_2 . We define as $\bar{\exists}_x r$ the sequence resulting by removing from $r \in \mathbf{M}$ all the information about the variable x . We distinguish two special classes of conditional traces. $r \in \mathbf{M}$ is said to be *self-sufficient* if the first condition is $(true, \emptyset)$ and, for each $t_i = (\eta_i^+, \eta_i^-) \rightarrow c_i$ and $t_{i+1} = (\eta_{i+1}^+, \eta_{i+1}^-) \rightarrow c_{i+1}$, $c_i \vdash \eta_{i+1}^+$ (each store satisfies the successive condition). Moreover, r is *x-self-sufficient* if $\bar{\exists}_{var \setminus \{x\}} r$ is self-sufficient. Thus, this definition demands that for self-sufficient conditional traces, no additional information (from other agents) is needed in order to *complete* the computation.⁵

The semantics definition is based on a semantics evaluation function $\mathcal{A}[[A]]_{\mathcal{I}}$ (Comini et al. 2013a) which, given an agent A and an interpretation \mathcal{I} , builds the conditional traces associated to A . The interpretation \mathcal{I} is a function which associates to each process symbol a set of conditional traces “modulo variance”. The semantics for a set of process declarations D is the fixpoint $\mathcal{F}[[D]] := lfp(\mathcal{D}[[D]])$ of the continuous immediate consequences operator $\mathcal{D}[[D]]_{\mathcal{I}}(p(\vec{x})) := \sqcup_{p(\vec{x}) \vdash A \in D} \mathcal{A}[[A]]_{\mathcal{I}}$. Proof of full abstraction w.r.t. the operational behavior of *tccp* is given in (Comini et al. 2013a).

Example 2.2 Consider the process declaration D_m of Example 2.1. Given an interpretation \mathcal{I} , the semantics of *master*(C, G) is graphically represented in Figure 2, where we

⁵ The set of all self-sufficient conditional traces can be considered as a generalization (using conditional states in place of stores) of the traditional strongest postcondition for semantics.

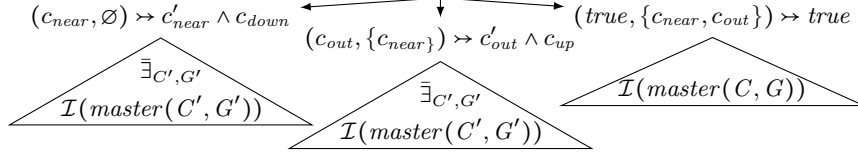


Fig. 2. Tree representation of $\mathcal{D}[\{D_m\}]_{\mathcal{I}}(\text{master}(C, G))$ of Example 2.2.

have used some shortcuts for characteristic constraints. Namely, $c_{\text{near}} := (C = [\text{near} \mid _])$, $c'_{\text{near}} := \exists_{C'}(C = [\text{near} \mid C'])$, $c_{\text{down}} := \exists_{G'}(G = [\text{down} \mid G'])$, $c_{\text{out}} := (C = [\text{out} \mid _])$, $c'_{\text{out}} := \exists_{C'}(C = [\text{out} \mid C'])$, $c_{\text{up}} := \exists_{G'}(G = [\text{up} \mid G'])$.

The branch on the left represents the computation when a *near* signal arrives. The first conditional state requires that c_{near} holds, thus the constraints c'_{near} and c_{down} are *concurrently* added to the store during that computational step. A recursive call is also concurrently invoked. Process calls do not modify the store when invoked, but they affect the store from the following time instant, which is graphically represented by the triangle labeled with the interpretation of the process. The branch in the middle is taken only if c_{out} is entailed and c_{near} is not entailed by the initial store (it occurs in the negative condition of the first conditional state in that branch). Finally, the branch on the right represents the case when both c_{near} and c_{out} are not entailed by the initial store. ■

3 Abstract semantics for *tccp* over **csLTL** formulas

In this section, we present a novel abstract semantics over formulas that approximates the small-step semantics described in Section 2 and, therefore, the small-step operational behavior of a *tccp* program. To this end, we first define an abstract domain of logic formulas which is a variation of the classical Linear Temporal Logic (Manna and Pnueli 1992). Following (Palamidessi and Valencia 2001; de Boer et al. 2001; de Boer et al. 2002; Valencia 2005), the idea is to replace atomic propositions by constraints of the underlying constraint system.

Definition 3.1 (csLTL formulas) *Given a cylindric constraint system \mathbf{C} , $c \in \mathbf{C}$ and $x \in \text{Var}$, formulas of the Constraint System Linear Temporal Logic over \mathbf{C} are:*

$$\phi ::= \text{true} \mid \text{false} \mid c \mid \neg \phi \mid \phi \wedge \phi \mid \exists_x \phi \mid \bigcirc \phi \mid \phi \mathcal{U} \phi.$$

$\text{csLTL}_{\mathbf{C}}$ is the set of all temporal formulas over \mathbf{C} (we omit \mathbf{C} if clear from the context).

true , false , \neg , \wedge , $\bigcirc \phi$, $\phi_1 \mathcal{U} \phi_2$ have the classical logical meaning. The atomic formula $c \in \mathbf{C}$ states that c has to be entailed by the current store. $\exists_x \phi$ is the existential quantification over the set of variables Var . As usual, we use $\phi_1 \dot{\vee} \phi_2$ as a shorthand for $\neg \phi_1 \wedge \neg \phi_2$; $\phi_1 \dot{\rightarrow} \phi_2$ for $\neg \phi_1 \dot{\vee} \phi_2$; $\phi_1 \leftrightarrow \phi_2$ for $\phi_1 \dot{\rightarrow} \phi_2 \wedge \phi_2 \dot{\rightarrow} \phi_1$; $\diamond \phi$ for $\text{true} \mathcal{U} \phi$ and $\square \phi$ for $\neg \diamond \neg \phi$. A *constraint formula* is an atomic formula c or its negation $\neg c$. Formulas $\bigcirc \phi$ and $\neg \bigcirc \phi$ are called *next* formulas. Constraint and next formulas are said to be *elementary* formulas. Finally, formulas of the form $\phi_1 \mathcal{U} \phi_2$, $\diamond \phi$ or $\neg(\square \phi)$ are called *eventualities*.

We define the abstract domain $\mathbb{F} := \text{csLTL} / \sim$ (i.e., the domain formed by **csLTL** formulas modulo logical equivalence) ordered by $\dot{\rightarrow}$. The algebraic lattice $(\mathbb{F}, \dot{\rightarrow}, \dot{\vee}, \dot{\wedge}, \text{true}, \text{false})$ is not complete, since both $\dot{\wedge}$ and $\dot{\vee}$ always exist just for finite sets of formulas.

The semantics of a temporal formula is typically defined in terms of an infinite sequence of states which validates it. Here we use conditional traces instead. As usually done in the context of temporal logics, we define the satisfaction relation \models only for infinite conditional traces. We implicitly transform finite traces (which end in \boxtimes) by replicating the last store infinite times.

Definition 3.2 *The semantics of $\phi \in \mathbb{F}$ is given by function $\gamma^\mathbb{F}: \mathbb{F} \rightarrow \mathbb{M}$ defined as $\gamma^\mathbb{F}(\phi) := \sqcup \{r \in \mathbf{M} \mid r \models \phi\}$, where, for each $\phi, \phi_1, \phi_2 \in \text{csLTL}$, $c \in \mathbf{C}$ and $r \in \mathbf{M}$, satisfaction relation \models is defined as:*

$$r \models \text{true} \quad \text{and} \quad r \not\models \text{false} \quad (3.1a)$$

$$(\eta^+, \eta^-) \rightsquigarrow d \cdot r' \models c \quad \text{iff} \quad \eta^+ \vdash c \quad (3.1b)$$

$$\text{stutt}(\eta^-) \cdot r' \models c \quad \text{iff} \quad \forall d^- \in \eta^- . c \not\vdash d^- \text{ and } r' \models c \quad (3.1c)$$

$$r \models \neg \phi \quad \text{iff} \quad r \not\models \phi \quad (3.1d)$$

$$r \models \phi_1 \wedge \phi_2 \quad \text{iff} \quad r \models \phi_1 \text{ and } r \models \phi_2 \quad (3.1e)$$

$$r \models \bigcirc \phi \quad \text{iff} \quad r^1 \models \phi^6 \quad (3.1f)$$

$$r \models \phi_1 \mathcal{U} \phi_2 \quad \text{iff} \quad \exists i \geq 1. \forall j < i. r^i \models \phi_2 \text{ and } r^j \models \phi_1 \quad (3.1g)$$

$$r \models \exists_x \phi \quad \text{iff exists } r' \text{ s.t. } \exists_x r' = \exists_x r, r' \text{ x-self-sufficient and } r' \models \phi \quad (3.1h)$$

We say that $\phi \in \mathbb{F}$ is a sound approximation of $R \in \mathbb{M}$ if $R \sqsubseteq \gamma^\mathbb{F}(\phi)$. ϕ is said to be satisfiable if there exists $r \in \mathbf{M}$ such that $r \models \phi$, while it is valid if, for all $r \in \mathbf{M}$, $r \models \phi$.

All the cases are fairly standard except (3.1b) and (3.1c). The conditional trace $r = (\eta^+, \eta^-) \rightsquigarrow d \cdot r'$ prescribes that η^+ is entailed by the current store, thus r models all the constraint formulas c such that $\eta^+ \vdash c$. We have to note that, by the monotonicity of the store of *tccp* computations, the positive conditions in conditional traces contains all the information previously added in the constraint store. Furthermore, by the definition of condition, since η^+ cannot be in contradiction with η^- , it holds that neither c is in contradiction with η^- . Thus, the conditional trace $\text{stutt}(\eta^-) \cdot r'$ models all the constraint formulas c that are not in contradiction with the set η^- and such that c holds in the continuation r' by monotonicity.

Lemma 3.3 *The function $\gamma^\mathbb{F}$ is monotonic, injective and \sqcap -distributive.*

3.1 csLTL Abstract Semantics

The technical core of our semantics definition is the **csLTL** agent semantics evaluation function $\hat{\mathcal{A}}[A]$ which, given an agent A and an interpretation $\hat{\mathcal{I}}$ (for the process symbols of A), builds a **csLTL** formula which is a sound approximation of the (concrete) behavior of A . In the sequel, we denote by $\mathbb{A}_\mathbf{C}^\Pi$ the set of agents and $\mathbb{D}_\mathbf{C}^\Pi$ the set of sets of process declarations built on signature Π and constraint system \mathbf{C} .

Definition 3.4 *Let $\mathbb{PC} := \{p(\vec{x}) \mid p \in \Pi, \vec{x} \text{ are distinct variables}\}$. An \mathbb{F} -interpretation is a function $\mathbb{PC} \rightarrow \mathbb{F}$ modulo variance⁷. Two functions $I, J: \mathbb{PC} \rightarrow \mathbb{F}$ are variants if for each $\pi \in \mathbb{PC}$ there exists a renaming ρ such that $(I\pi)\rho = J(\pi\rho)$. The semantic domain $\mathbb{I}_\mathbb{F}$ is the set of all \mathbb{F} -interpretations ordered by the point-wise extension of $\dot{\rightarrow}$.*

⁶ r^k denotes the sub-sequence of r starting from state k .

⁷ i.e., a family of elements of \mathbb{F} , indexed by \mathbb{PC} , modulo variance.

Definition 3.5 (csLTL Semantics) Given $A \in \mathbb{A}_{\mathbf{C}}^{\Pi}$ and $\hat{\mathcal{I}} \in \mathbb{I}_{\mathbb{F}}$, we define the csLTL semantics evaluation $\hat{\mathcal{A}}[\![A]\!]_{\hat{\mathcal{I}}}$ by structural induction as follows.

$$\begin{aligned}\hat{\mathcal{A}}[\![\text{skip}]\!]_{\hat{\mathcal{I}}} &:= \text{true} & \hat{\mathcal{A}}[\![A_1 \parallel A_2]\!]_{\hat{\mathcal{I}}} &:= \hat{\mathcal{A}}[\![A_1]\!]_{\hat{\mathcal{I}}} \wedge \hat{\mathcal{A}}[\![A_2]\!]_{\hat{\mathcal{I}}} \\ \hat{\mathcal{A}}[\![\text{tell}(c)]\!]_{\hat{\mathcal{I}}} &:= \bigcirc c & \hat{\mathcal{A}}[\![\exists x A]\!]_{\hat{\mathcal{I}}} &:= \dot{\exists}_x \hat{\mathcal{A}}[\![A]\!]_{\hat{\mathcal{I}}} & \hat{\mathcal{A}}[\![p(\vec{x})]\!]_{\hat{\mathcal{I}}} &:= \bigcirc \hat{\mathcal{I}}(p(\vec{x})) \\ \hat{\mathcal{A}}[\![\sum_{i=1}^n \text{ask}(c_i) \rightarrow A_i]\!]_{\hat{\mathcal{I}}} &:= \square(\dot{\wedge}_{i=1}^n \dot{\neg} c_i) \dot{\vee} \left((\dot{\wedge}_{i=1}^n \dot{\neg} c_i) \mathcal{U} \dot{\vee}_{i=1}^n (c_i \dot{\wedge} \bigcirc \hat{\mathcal{A}}[\![A_i]\!]_{\hat{\mathcal{I}}}) \right) \\ \hat{\mathcal{A}}[\![\text{now } c \text{ then } A_1 \text{ else } A_2]\!]_{\hat{\mathcal{I}}} &:= (c \dot{\wedge} \hat{\mathcal{A}}[\![A_1]\!]_{\hat{\mathcal{I}}}) \dot{\vee} (\dot{\neg} c \dot{\wedge} \hat{\mathcal{A}}[\![A_2]\!]_{\hat{\mathcal{I}}})\end{aligned}$$

Given $D \in \mathbb{D}_{\mathbf{C}}^{\Pi}$ we define the immediate consequence operator $\hat{\mathcal{D}}[\![D]\!]: \mathbb{I}_{\mathbb{F}} \rightarrow \mathbb{I}_{\mathbb{F}}$ as

$$\hat{\mathcal{D}}[\![D]\!]_{\hat{\mathcal{I}}}(p(\vec{x})) := \dot{\vee} \{ \hat{\mathcal{A}}[\![A]\!]_{\hat{\mathcal{I}}} \mid p(\vec{x}) :- A \in D \}$$

We have that $\hat{\mathcal{A}}$ is a sound approximation of \mathcal{A} and $\hat{\mathcal{D}}$ is a sound approximation of \mathcal{D} .

Theorem 3.6 (Correctness of $\hat{\mathcal{A}}$ and $\hat{\mathcal{D}}$) Let $A \in \mathbb{A}_{\mathbf{C}}^{\Pi}$, $D \in \mathbb{D}_{\mathbf{C}}^{\Pi}$ and $\hat{\mathcal{I}} \in \mathbb{I}_{\mathbb{F}}$. Then, $\mathcal{A}[\![A]\!]_{\gamma^{\mathbb{F}}(\hat{\mathcal{I}})} \sqsubseteq \gamma^{\mathbb{F}}(\hat{\mathcal{A}}[\![A]\!]_{\hat{\mathcal{I}}})$ and $\mathcal{D}[\![D]\!]_{\gamma^{\mathbb{F}}(\hat{\mathcal{I}})} \sqsubseteq \gamma^{\mathbb{F}}(\hat{\mathcal{D}}[\![D]\!]_{\hat{\mathcal{I}}})$.

Example 3.7 Consider the process declaration D_m of Example 2.1 and let us use \bigcirc^n to abbreviate the repetition of \bigcirc n -times. Given $\hat{\mathcal{I}} \in \mathbb{I}_{\mathbb{F}}$, with Definition 3.5 we compute

$$\phi_M(\hat{\mathcal{I}}) := \hat{\mathcal{D}}[\![\{D_m\}]\!]_{\hat{\mathcal{I}}}(\text{master}(C, G)) = \phi_{\text{near}}(\hat{\mathcal{I}}) \dot{\vee} \phi_{\text{out}}(\hat{\mathcal{I}}) \dot{\vee} \phi_{\text{cwait}}(\hat{\mathcal{I}})$$

where

$$\begin{aligned}\phi_{\text{near}}(\hat{\mathcal{I}}) &= \dot{\exists}_{C', G'} (C = [\text{near} \mid _] \dot{\wedge} \bigcirc C = [\text{near} \mid C'] \dot{\wedge} \bigcirc G = [\text{down} \mid G'] \dot{\wedge} \bigcirc \hat{\mathcal{I}}(\text{master}(C', G'))) \\ \phi_{\text{out}}(\hat{\mathcal{I}}) &= \dot{\exists}_{C', G'} (\dot{\neg}(C = [\text{near} \mid _]) \dot{\wedge} \bigcirc C = [\text{out} \mid C'] \dot{\wedge} \\ &\quad C = [\text{out} \mid _] \dot{\wedge} \bigcirc G = [\text{up} \mid G'] \dot{\wedge} \bigcirc \hat{\mathcal{I}}(\text{master}(C', G'))) \\ \phi_{\text{cwait}}(\hat{\mathcal{I}}) &= \dot{\neg}(C = [\text{near} \mid _]) \dot{\wedge} \dot{\neg}(C = [\text{out} \mid _]) \dot{\wedge} \bigcirc \hat{\mathcal{I}}(\text{master}(C, G))\end{aligned}$$

The three disjuncts of $\phi_M(\hat{\mathcal{I}})$ match the three possible behaviors of $\text{master}(C, G)$: when signal *near* is emitted by the train, when *out* is emitted, and when no signal arrives. ■

4 Abstract diagnosis of *tccp* with csLTL formulas

Since \mathbb{F} is not a complete lattice, it is impossible to find for the function $\gamma^{\mathbb{F}}$ an adjoint function α which forms a Galois Connection $\langle \alpha, \gamma \rangle$, and therefore we cannot use the abstract diagnosis framework for *tccp* defined in (Comini et al. 2011). Thus, we propose in this section a new weaker version of abstract diagnosis that works on \mathbb{F} ⁸.

Given a set of declarations D and $\hat{\mathcal{S}} \in \mathbb{I}_{\mathbb{F}}$, which is the specification of the abstract intended behavior of D over \mathbb{F} , we say that

1. D is (abstractly) *partially correct* w.r.t. $\hat{\mathcal{S}}$ if $\mathcal{F}[\![D]\!] \sqsubseteq \gamma^{\mathbb{F}}(\hat{\mathcal{S}})$.
2. D is (abstractly) *complete* w.r.t. $\hat{\mathcal{S}}$ if $\gamma^{\mathbb{F}}(\hat{\mathcal{S}}) \sqsubseteq \mathcal{F}[\![D]\!]$.

The differences between $\mathcal{F}[\![D]\!]$ and $\gamma^{\mathbb{F}}(\hat{\mathcal{S}})$ are usually called *symptoms*. Many of the

⁸ Actually, the proposal is defined using just $\gamma^{\mathbb{F}}$ only for the sake of simplicity. It could easily be defined parametrically w.r.t. a suitable family of concretization functions.

symptoms are just a consequence of some “originating” ones, those which are the direct consequence of errors. The *abstract diagnosis* determines exactly the “originating” symptoms and, in the case of incorrectness, the faulty process declarations in D . This is captured by the definitions of *abstractly incorrect process declaration* and *abstract uncovered element*.⁹

Definition 4.1 Let $D \in \mathbb{D}_C^\Pi$, R a process declaration for process p , $\phi_t \in \mathbb{F}$ and $\hat{S} \in \mathbb{I}_F$.

- R is abstractly incorrect w.r.t. \hat{S} (on testimony ϕ_t) if $\phi_t \dot{\rightarrow} \hat{\mathcal{D}}[\{R\}]_{\hat{S}}(p(\vec{x}))$ and $\phi_t \wedge \hat{S}(p(\vec{x})) = \text{false}$.
- ϕ_t is an uncovered element for $p(\vec{x})$ w.r.t. \hat{S} if $\phi_t \dot{\rightarrow} \hat{S}(p(\vec{x}))$ and $\phi_t \wedge \hat{\mathcal{D}}[D]_{\hat{S}}(p(\vec{x})) = \text{false}$.

Informally, R is abstractly incorrect if it derives a wrong abstract element ϕ_t from the intended semantics. Dually, ϕ_t is uncovered if the declarations cannot derive it from the intended semantics.

Theorem 4.2 Let $D \in \mathbb{D}_C^\Pi$ and $\hat{S} \in \mathbb{I}_F$. (1) If there are no abstractly incorrect process declarations in D (i.e., $\hat{\mathcal{D}}[D]_{\hat{S}} \dot{\rightarrow} \hat{S}$), then D is partially correct w.r.t. \hat{S} . (2) If D is partially correct w.r.t. \hat{S} and D has abstract uncovered elements then D is not complete.

Absence of abstractly incorrect declarations is a sufficient condition for partial correctness, but it is not necessary. Because of the approximation, it can happen that a (concretely) correct declaration is abstractly incorrect. Hence, abstract incorrect declarations are in general just a warning about a possible source of errors. However, an abstract correct declaration cannot contain an error; thus, no (manual) inspection is needed for declarations which are not abstractly incorrect. Moreover, as shown by the following theorem, all concrete errors—that are “visible”—are indeed detected, as they lead to an abstract incorrectness or abstract uncovered. Intuitively, a concrete error is *visible* if we can express a formula ϕ whose concretization reveals the error (i.e., if the logic is expressive enough).

Theorem 4.3 Let R be a process declaration for $p(\vec{x})$, \mathcal{S} a concrete specification and \hat{S} a sound approximation for \mathcal{S} (i.e., $\mathcal{S} \sqsubseteq \gamma^F(\hat{S})$). (1) If $\mathcal{D}[\{R\}]_{\mathcal{S}} \not\sqsubseteq \gamma^F(\hat{S})$ and it exists ϕ_t such that $\gamma^F(\phi_t) \sqsubseteq \mathcal{D}[\{R\}]_{\mathcal{S}}(p(\vec{x}))$ and $\phi_t \wedge \hat{S}(p(\vec{x})) = \text{false}$, then R is abstractly incorrect w.r.t. \hat{S} (on testimony ϕ_t). (2) If there exists an abstract uncovered element ϕ w.r.t. \hat{S} , then there exists $r \in \gamma^F(\phi)$ such that $r \notin \mathcal{D}[\{R\}]_{\mathcal{S}}(p(\vec{x}))$.

Point 2 says that the concrete error has an abstract symptom which is not hidden by the approximation on \hat{S} and, moreover, there exists a formula ϕ_t which can express it.

In the following examples, we borrow from (Alpuente et al. 2006) the notation for *last entailed value* of a stream: $X \dot{=} c$ holds if the last instantiated value in the stream X is c .

Example 4.4 We verify (for Example 2.1) that each time a *near* signal arrives from a train, the order *down* is sent to a gate process.¹⁰ To model this property, we define the specification (of the property) $\hat{\mathcal{S}}_{\text{down}}$ as

$$\phi_{\text{ordersent}} := \hat{\mathcal{S}}_{\text{down}}(\text{master}(C, G)) := \Box(C \dot{=} \text{near} \rightarrow \Diamond(G \dot{=} \text{down}))$$

⁹ It is worth noticing that although the notions defined in this section are similar to those defined for the standard approach, the formal definitions and proofs are different due to the weaker framework.

¹⁰ A more interesting property, namely that, in addition, the gate is eventually down, is verified in (Comini et al. 2014). Here we have simplified the property due to space limitations.

To check whether the program implies the specification ($\hat{\mathcal{D}}[\{D_m\}]_{\hat{\mathcal{S}}_{down}} \dot{\rightarrow} \hat{\mathcal{S}}_{down}$) we have to check if $\phi_M(\hat{\mathcal{S}}_{down}) \dot{\rightarrow} \phi_{ordersent}$ (where $\phi_M(\cdot)$ is defined in Example 3.7). Each of the three disjuncts of $\phi_M(\hat{\mathcal{S}}_{down})$ implies $\phi_{ordersent}$. Thus, by Theorem 4.2, D_m is partially correct w.r.t. $\hat{\mathcal{S}}_{down}$. ■

When the check of a process declaration R against a specification S fails, our method reports that R is not partially correct w.r.t. S . If this occurs, the formula testimony for the possible incorrectness gives useful information to fix the process declaration or check whether it corresponds to a false positive.

Example 4.5 Now we show how our technique detects an error in a buggy set of declarations. We remove instruction `tell($G = [up \mid G']$)` in the process declaration D_m (of Example 2.1). To avoid misunderstandings, we call the modified process $master'$ and let R be the new process declaration.

We aim to verify that the order up is sent whenever the signal out is received:

$$\phi := \hat{\mathcal{S}}_{up}(master'(C, G)) := \Box((C \dot{=} out) \dot{\rightarrow} \Diamond(G \dot{=} up))$$

We need to compute the (one step) semantics for the (buggy version of the) process:

$$\phi' := \hat{\mathcal{D}}[\{R\}]_{\hat{\mathcal{S}}_{up}}(master'(C, G)) = \phi'_{near} \dot{\vee} \phi'_{out} \dot{\vee} \phi'_{cwait}$$

where

$$\begin{aligned} \phi'_{near} &:= \exists_{C', G'} (C = [near \mid _]\dot{\wedge} \bigcirc C = [near \mid C'] \dot{\wedge} \bigcirc G = [down \mid G'] \dot{\wedge} \bigcirc \hat{\mathcal{S}}_{up}(master'(C', G'))) \\ \phi'_{out} &:= \exists_{C', G'} (\neg(C = [near \mid _]) \dot{\wedge} C = [out \mid _]\dot{\wedge} \bigcirc(C = [out \mid C'] \dot{\wedge} \bigcirc \hat{\mathcal{S}}_{up}(master'(C', G')))) \\ \phi'_{cwait} &:= \neg(C = [near \mid _]) \dot{\wedge} \neg(C = [out \mid _]) \dot{\wedge} \bigcirc \hat{\mathcal{S}}_{up}(master'(C, G)) \end{aligned}$$

We detect an incorrectness of R (in $master'$ process) w.r.t. $\hat{\mathcal{S}}_{up}$ on testimony ϕ'_{out} since $\phi'_{out} \dot{\rightarrow} \phi'$ and $\phi'_{out} \dot{\wedge} \phi = false$. The testimony suggests that on channel C we have out signal but we do not see the corresponding up signal on channel G . ■

Our technique behaves negatively for sets of declarations D where $\hat{\mathcal{D}}[D]$ has more than one fixpoint. This happens with programs with loops that do not produce contributes at all (which are in some sense non meaningful programs). In such situations, we can have that the actual behavior does not model a specification $\hat{\mathcal{S}}$ which is a non-least fixpoint of $\hat{\mathcal{D}}[D]$, but, since $\hat{\mathcal{S}}$ is a fixpoint, we do not detect the abstractly incorrect declaration, as shown by the following example.

Example 4.6 (Pathological cases) Let $D_p := \{q(y) :- \text{now } y = 1 \text{ then } q(y) \text{ else } q(y)\}$ and $\hat{\mathcal{S}}_p(q(y)) := \Diamond(y = 1)$ be the specification. Then, we compute $\hat{\mathcal{D}}[D_p]_{\hat{\mathcal{S}}_p}(q(y)) = (y = 1 \dot{\wedge} \Diamond(y = 1)) \dot{\vee} (\neg(y = 1) \dot{\wedge} \Diamond(y = 1))$. We can see that $\hat{\mathcal{D}}[D_p]_{\hat{\mathcal{S}}_p} \dot{\rightarrow} \Diamond(y = 1)$, thus D_p is partially correct w.r.t. $\hat{\mathcal{S}}_p$. However, $y = 1$ is not explicitly added by the process. ■

Note that, if $\hat{\mathcal{S}}(p(\vec{x}))$ is assumed to hold for each process $p(\vec{x})$ defined in D and $\hat{\mathcal{D}}[D]_{\hat{\mathcal{S}}} \dot{\rightarrow} \hat{\mathcal{S}}$, then $\mathcal{F}[D]$ satisfies $\hat{\mathcal{S}}$.

4.1 An automatic decision procedure for csLTL

In order to make our abstract diagnosis approach effective, we have defined an automatic decision procedure to check the validity of the formulas involved in Definition 4.1 (of the

Table 1. α - and β -formulas rules.

	α	$A(\alpha)$		β	$B_1(\beta)$	$B_2(\beta)$
R1	$\neg\neg\phi$	$\{\phi\}$	R4	$\neg(\phi_1 \wedge \phi_2)$	$\{\neg\phi_1\}$	$\{\neg\phi_2\}$
R2	$\phi_1 \wedge \phi_2$	$\{\phi_1, \phi_2\}$	R5	$\neg(\phi_1 \mathcal{U} \phi_2)$	$\{\neg\phi_1, \neg\phi_2\}$	$\{\phi_1, \neg\phi_2, \neg\bigcirc(\phi_1 \mathcal{U} \phi_2)\}$
R3	$\neg\bigcirc\phi$	$\{\bigcirc\neg\phi\}$	R6	$\phi_1 \mathcal{U} \phi_2$	$\{\phi_2\}$	$\{\phi_1, \neg\phi_2, \bigcirc((\Gamma^* \wedge \phi_1) \mathcal{U} \phi_2)\}$

form $\psi \dot{\rightarrow} \phi$ with $\phi = \hat{S}(p(\vec{x}))$ and $\psi = \hat{D}[\llbracket D \rrbracket_{\mathcal{S}}(p(\vec{x}))]$. We adapt to **csLTL** the tableau construction for Propositional LTL of (Gaintzarain et al. 2008; Gaintzarain et al. 2009). (Comini et al. 2013b) contains a preliminary version of the method.

Intuitively, a tableau consists of a tree whose nodes are labeled with sets of formulas. The root is labeled with the set of formulas which has to be checked for satisfiability. Branches are built according to rules defined on the syntax of formulas (see Table 1 defining α and β formulas). The basic idea is that a formula from a node is selected and, depending on its form, a rule of Table 1 is applied. β formulas generate a bifurcation on the tree and there are specific rules for next and existential quantification formulas.

If all branches of the tree are *closed* (Definition 4.8), then the formula has no models. Otherwise, we can obtain a model from the *open* branches.

Definition 4.7 (csLTL tableau) A *csLTL tableau* for a finite set of formulas Φ is a tuple $\mathcal{T}_\Phi = (\text{Nodes}, n_\Phi, L, B, R)$ such that:

1. *Nodes* is a finite non-empty set of nodes;
2. $n_\Phi \in \text{Nodes}$ is the initial node;
3. $L : \text{Nodes} \rightarrow \wp(\text{csLTL})$ is the labeling function that associates to each node the formulas which are true in that node; the initial node is labeled with Φ ;
4. B is the set of branches such that exactly one of the following points holds for every branch $b = n_0, \dots, n_i, n_{i+1}, \dots, n_l \in B$ and every $0 \leq i < l$:
 - (a) for an α -formula $\alpha \in L(n_i)$, $L(n_{i+1}) = \{A(\alpha)\} \cup L(n_i) \setminus \{\alpha\}$;
 - (b) for a β -formula $\beta \in L(n_i)$, $L(n_{i+1}) = \{B_1(\beta)\} \cup L(n_i) \setminus \{\beta\}$ and there exists another branch in B of the form $b' = n_0, \dots, n_i, n'_{i+1}, \dots, n'_k$ such that $L(n'_{i+1}) = \{B_2(\beta)\} \cup L(n_i) \setminus \{\beta\}$;
 - (c) for an existential quantified formula $\exists_x \phi' \in L(n_i)$, $L(n_{i+1}) = \{\phi''\} \cup L(n_i) \setminus \{\exists_x \phi'\}$ where $\phi'' := \phi'[y/x]$ with y fresh variable;
 - (d) in case $L(n_i)$ is a set formed only by elementary formulas, $L(n_{i+1}) = \text{next}(L(n_i))$, where $\text{next}(\Phi) := \{\phi \mid \bigcirc\phi \in \Phi\} \cup \{\neg\phi \mid \neg\bigcirc\phi \in \Phi\} \cup (\Phi \cap \mathbf{C})$.

Rules 4a and 4b are standard, replacing α and β -formulas with one or two formulas according to the matching pattern of rules in Table 1, except for Rule R6 that uses the so-called context Γ^* , which is defined in the following. The *next* operator used in Rule 4d is different from the corresponding one of PLTL since it also preserves the constraint formulas. This is needed for guaranteeing correctness in the particular setting of *tccp* where the store is monotonic. Finally, Rule 4c is specific for the \exists case: \exists_x is removed after renaming x with a fresh variable¹¹.

Definition 4.8 A node in the tableau is *inconsistent* if it contains a couple of formulas $\phi, \neg\phi$, or the formula *false*, or a constraint formula $\neg c'$ such that the merge c of all the

¹¹ The **csLTL** existential quantification does not correspond to the one of FO logic. It serves to model local variables, and $\exists_x \phi$ can be seen just as ϕ where the information about x is local.

(positive) constraint formulas c_1, \dots, c_n in the node (i.e., $c := c_1 \otimes \dots \otimes c_n$) is such that $c \vdash c'$. A branch is closed if it contains an inconsistent node.

The last condition for inconsistency of a node is particular to the *ccp* context.

We now describe the algorithm that automatically builds the **csLTL** tableau for a given set of formulas Φ (see (Comini et al. 2014) for the pseudocode). The construction consists in selecting at each step a branch that can be extended by using α or β rules or \exists elimination. When none of these can be applied, the **next** operator is used to pass to the next *stage*. When dealing with eventualities, to determine the context Γ^* in Rule **R6**, it is necessary to *distinguish* the eventuality that is being unfolded in the path. Given a node n and $\phi \in L(n)$, $\Gamma := L(n) \setminus \{\phi\}$. Then, when Rule **R6** is applied to a *distinguished* eventuality, we set $\Gamma^* := \dot{\bigvee}_{\gamma \in \Gamma} \dot{\neg} \gamma$; otherwise $\Gamma^* := \text{true}$. The use of contexts is the mechanism to detect the loops that allows one to mark branches containing eventuality formulas as *open* or to generate inconsistent nodes and mark branches as *closed*. A node is marked as *closed* when it is inconsistent while is marked as *open* when (1) it is the last node of the branch and contains just constraint formulas or (2) the branch is cyclic and all the eventualities in the cycle have been already distinguished.

In order to ensure termination of the algorithm, it is necessary to use a *fair* strategy to distinguish eventualities, in the sense that every eventuality in an open branch must be distinguished at some point. This assumption and the fact that, given a finite set of initial formulas, there exists only a finite set of possible labels in a systematic tableau, imply termination of the tableau construction. Moreover, the constructed tableau is sound and complete. Therefore, to check the validity of a formula of the form $\psi \dot{\rightarrow} \phi$, with $\phi = \hat{S}(p(\vec{x}))$ and $\psi = \hat{\mathcal{D}}[\![D]\!]_{\hat{S}}(p(\vec{x}))$, we just have to build the tableau for its negation $\mathcal{T}_{\neg(\psi \dot{\rightarrow} \phi)}$ and check if it is closed or not. If it is, we have that D is abstractly correct. Otherwise, we can extract from $\mathcal{T}_{\neg(\psi \dot{\rightarrow} \phi)}$ an explicit testimony φ of the abstract incorrectness of D .

The construction of $\psi = \hat{\mathcal{D}}[\![D]\!]_{\hat{S}}(p(\vec{x}))$ is linear in the size of D . The systematic tableau construction of $\neg(\psi \dot{\rightarrow} \phi)$ (from what said in (Gaintzarain et al. 2009)) has worst case $O(2^{O(2^{|\neg(\psi \dot{\rightarrow} \phi)|})})$. However, we believe that such bound for the worst-case asymptotic behavior is quite meaningless in this context, since it is not very realistic to think that the formulas of the specification should grow much (big formulas are difficult to comprehend and in real situations people would hardly try even to imagine them). Moreover, note that tableau explosion is due to nesting of eventualities and in practice really few eventualities are used in specifications. Therefore, in real situations, we do not expect that (extremely) big tableaux will be built.

5 Related Work

A Constraint Linear Temporal Logic is defined in (Valencia 2005) for the verification of a different timed concurrent language, called *ntcc*, which shares with *tccp* the concurrent constraint nature and the non-monotonic behavior. The restricted negation fragment of this logic, where negation is only allowed for state formulas, is shown to be decidable. However, no efficient decision procedure is given (apart from the proof itself). Moreover, the verification results are given for the locally-independent fragment of *ntcc*, which avoids the non-monotonicity of the original language. In contrast, in this work, we address the problem of checking temporal properties for the full *tccp* language.

Some model-checking techniques have been defined for *tccp* in the past (Falaschi and Villanueva 2006; Alpuente et al. 2005a; Alpuente et al. 2005b; Falaschi et al. 2001). It is worth noting that the notions of correctness and completeness in these works are defined in terms of $\mathcal{F}[[D]]$, i.e., in terms of the concrete semantics, and therefore their check requires a (potentially infinite) fixpoint computation. In contrast, the notions of abstractly incorrect declarations and abstract uncovered elements are defined in terms of *just one* application of $\hat{\mathcal{D}}[[D]]$ to $\hat{\mathcal{S}}$. Moreover, since $\hat{\mathcal{D}}[[D]]$ is defined compositionally, all the checks are defined on each process declaration in isolation. Hence, our proposal can be used with partial sets of declarations. When a property is falsified, model checking provides a counterexample in terms of an erroneous execution trace, leaving to the user the problem of locating the source of the bug. On the contrary, we identify the faulty process declaration.

In (Falaschi et al. 2007), a first approach to the declarative debugging of a *ccp* language is presented. However, it does not cover the particular extra difficulty of the non-monotonicity behavior, common to all timed concurrent constraint languages. This makes our approach significantly different. Moreover, although they propose the use of LTL for the specification of properties, their formulation, based on the depth-k concretization function, complicates the task of having an efficient implementation.

Finally, this proposal clearly relates to the abstract diagnosis framework for *tccp* defined for Galois Insertions (Comini et al. 2011). That work can compete with the precision of model checking, but its main drawback is the fact that the abstract domain did not allow to specify temporal properties in a compact way. In fact, specifications consisted of sets of *abstract conditional traces*. Thus, specifications were big and unnatural to be written. The use of temporal logic in this proposal certainly overcomes this problem.

6 Conclusion and Future Work

We have defined an abstract semantics for *tccp* based on the domain of a linear temporal logic with constraints. The semantics is correct w.r.t. the behavior of the language.

By using this abstract semantics, we have defined a method to validate csLTL formulas for *tccp* sets of declarations. Since the abstract semantics cannot be defined by means of a Galois Connection, we cannot use the abstract diagnosis framework for *tccp* defined in (Comini et al. 2011), thus we devised (from scratch) a weak version of the abstract diagnosis framework based only on a concretization function γ . It works by applying $\hat{\mathcal{D}}[[D]]$ to the abstract specification and then by checking the validity of the resulting implications (whether that computation implies the abstract specification). The computational cost depends essentially on the cost of that check of the implication.

We have also presented an automatic decision procedure for the csLTL logic, thus we can effectively check the validity of that implication. We are currently finishing to implement a proof of concept tool, which is available online at URL <http://safe-tools.dsic.upv.es/tadi/>, that realizes the proposed instance. Then we would be able to compare with other tools and assess the “real life” goodness of our proposal.

In the future, we also plan to explore other instances of the method based on logics for which decision procedures or (semi)automatic tools exists. This proposal can also be immediately adapted to other concurrent (non-monotonic) languages (like *tcc* and *ntcc*) once a suitable fully abstract semantics has been developed.

References

- ALPUENTE, M., FALASCHI, M., AND VILLANUEVA, A. 2005a. A Symbolic Model Checker for tccp Programs. In *First International Workshop on Rapid Integration of Software Engineering Techniques (RISE 2004), Revised Selected Papers*. Lecture Notes in Computer Science, vol. 3475. Springer-Verlag, 45–56.
- ALPUENTE, M., GALLARDO, M., PIMENTEL, E., AND VILLANUEVA, A. 2005b. A Semantic Framework for the Abstract Model Checking of tccp Programs. *Theoretical Computer Science* 346, 1, 58–95.
- ALPUENTE, M., GALLARDO, M., PIMENTEL, E., AND VILLANUEVA, A. 2006. Verifying Real-Time Properties of tccp Programs. *Journal of Universal Computer Science* 12, 11, 1551–1573.
- CLARKE, E. M. AND EMERSON, E. A. 1981. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, D. Kozen, Ed. Lecture Notes in Computer Science, vol. 131. Springer, 52–71.
- COMINI, M., TITOLO, L., AND VILLANUEVA, A. 2011. Abstract Diagnosis for Timed Concurrent Constraint programs. *Theory and Practice of Logic Programming* 11, 4-5, 487–502.
- COMINI, M., TITOLO, L., AND VILLANUEVA, A. 2013a. A Condensed Goal-Independent Bottom-Up Fixpoint Modeling the Behavior of tccp. Tech. rep., DSIC, Universitat Politècnica de València. Available at <http://riunet.upv.es/handle/10251/34328>.
- COMINI, M., TITOLO, L., AND VILLANUEVA, A. 2013b. Towards an Effective Decision Procedure for LTL formulas with Constraints. In 23rd Workshop on Logic-based methods in Programming Environments (WLPE 2013). *CoRR abs/1308.2055*.
- COMINI, M., TITOLO, L., AND VILLANUEVA, A. 2014. Abstract Diagnosis for tccp using a Linear Temporal Logic. Tech. rep., Universitat Politècnica de València. Available at <http://riunet.upv.es/handle/10251/8351>.
- DE BOER, F. S., GABBRIELLI, M., AND MEO, M. C. 2000. A Timed Concurrent Constraint Language. *Information and Computation* 161, 1, 45–83.
- DE BOER, F. S., GABBRIELLI, M., AND MEO, M. C. 2001. A Temporal Logic for Reasoning about Timed Concurrent Constraint Programs. In *TIME '01: Proceedings of the Eighth International Symposium on Temporal Representation and Reasoning (TIME'01)*. IEEE Computer Society, Washington, DC, USA, 227.
- DE BOER, F. S., GABBRIELLI, M., AND MEO, M. C. 2002. Proving correctness of Timed Concurrent Constraint Programs. *CoRR cs.LO/0208042*.
- FALASCHI, M., OLARTE, C., PALAMIDESSI, C., AND VALENCIA, F. D. 2007. Declarative Diagnosis of Temporal Concurrent Constraint Programs. In *Logic Programming, 23rd International Conference, ICLP 2007, Proceedings*, V. Dahl and I. Niemelä, Eds. Lecture Notes in Computer Science, vol. 4670. Springer-Verlag, 271–285.
- FALASCHI, M., POLICRITI, A., AND VILLANUEVA, A. 2001. Modeling concurrent systems specified in a temporal concurrent constraint language-I. *Electronic Notes in Theoretical Computer Science* 48, 197–210.
- FALASCHI, M. AND VILLANUEVA, A. 2006. Automatic verification of timed concurrent constraint programs. *Theory and Practice of Logic Programming* 6, 3, 265–300.
- GAINTZARAIN, J., HERMO, M., LUCIO, P., AND NAVARRO, M. 2008. Systematic semantic tableaux for PLTL. *Electronic Notes in Theoretical Computer Science* 206, 59–73.
- GAINTZARAIN, J., HERMO, M., LUCIO, P., NAVARRO, M., AND OREJAS, F. 2009. Dual Systems of Tableaux and Sequents for PLTL. *The Journal of Logic and Algebraic Programming* 78, 8, 701–722.
- MANNA, Z. AND PNUCLI, A. 1992. *The temporal logic of reactive and concurrent systems - specification*. Springer.
- PALAMIDESSI, C. AND VALENCIA, F. D. 2001. A Temporal Concurrent Constraint Programming Calculus. In *7th International Conference on Principles and Practice of Constraint Programming (CP'01)*. Lecture Notes in Computer Science, vol. 2239. Springer, 302–316.

- QUEILLE, J. P. AND SIFAKIS, J. 1982. Specification and verification of concurrent systems in CESAR. In *Symposium on Programming*, M. Dezani-Ciancaglini and U. Montanari, Eds. Lecture Notes in Computer Science, vol. 137. Springer, 337–351.
- SARASWAT, V. A. 1989. Concurrent Constraint Programming Languages. Ph.D. thesis, Carnegie-Mellon University.
- SARASWAT, V. A. 1993. *Concurrent Constraint Programming*. The MIT Press, Cambridge, Mass.
- VALENCIA, F. D. 2005. Decidability of infinite-state timed CCP processes and first-order LTL. *Theoretical Computer Science* 330, 3, 577–607.